Memento: Making Sliding Windows Efficient for Heavy Hitters

Ran Ben Basat, Gil Einziger, Isaac Keslassy, Ariel Orda, Shay Vargaftik, and Erez Waisbard

***Abstract*—Cloud operators require timely identification of Heavy Hitters (HH) and Hierarchical Heavy Hitters (HHH) for applications such as load balancing, traffic engineering, and attack mitigation. However, existing techniques are slow in detecting new heavy hitters.**

In this paper, we present the case for identifying heavy hitters through *sliding windows*. Sliding windows are quicker and more accurate to detect new heavy hitters than current interval-based methods, but to date had no practical algorithms. Accordingly, we introduce, design, and analyze the *Memento* family of sliding window algorithms for the HH and HHH problems in the single-device and network-wide settings. We use extensive evaluations to show that our single-device solutions are orders of magnitude faster than existing sliding window techniques and comparable in speed to state-of-the-art non-windowed sampling based technique.

Furthermore, we exemplify our network-wide HHH detection capabilities on a realistic testbed. To that end, we implemented Memento as an open-source extension to the popular HAProxy cloud load-balancer. In our evaluations, using an HTTP flood by 50 subnets, our network-wide approach detected the new subnets faster and reduced the number of undetected flood requests by up to $37\times$ compared to the alternatives.

I. Introduction

Cloud operators require fast and accurate single-device and network-wide detection of *Heavy Hitters (HH)* (most frequent flows) and of *Hierarchical Heavy Hitters (HHH)* (most frequent subnets) to attain real-time visibility of their traffic. These capabilities are essential building blocks in network functions that are key to network softwarization, such as load balancing [7], [26], [31], traffic engineering [16] and attack mitigation [37], [35], [41].

Quickly identifying changes in the HH and HHH is a key challenge [39] and can have a dramatic impact on the performance, reliability, and security of network components. For example, faster detection of HH flows allows load balancing and traffic engineering solutions to respond to traffic spikes swiftly. For attack mitigation systems, quicker and more accurate detection of HHH subnets means that less attack traffic reaches the victim. Faster detection is particularly important for fighting *Distributed Denial of Service (DDoS)* attacks on cloud services that grow with the increase in connected devices (*i.e.,* Internet of Things) [24], [27].

Contributions. We first show that *sliding windows* are more accurate than interval-based measurements. Despite their merits, existing sliding window algorithms were considered “markedly slower and less space-efficient in practice”, to quote [32].

We introduce the *Memento* family of four algorithms for the fundamental HH and HHH problems in both single-device and network-wide measurements. We rigorously analyze them, establish accuracy guarantees and maximize accuracy given a per-packet control bandwidth budget. Using extensive evaluations on real packet traces, we show that the Memento algorithms achieve speedups of up to $14\times$ in HH and up to $273\times$ in HHH when compared to existing sliding-window solutions. These speedups come from utilizing our sampling-based approach at the cost of weaker approximation guarantees. We further show that they match the speed of the fastest interval-based algorithm [10].

Next, we implement a proof-of-concept network-wide HH and HHH measurement system. We evaluate the achievable accuracy when limiting the communication overhead for reporting results to the controller. We experiment with different reporting strategies and evaluate their impact on accuracy. We create an HTTP flood attack from 50 subnets and show that the detection time is near-optimal while using a bandwidth budget of 1 byte per packet. For the same budget, our methods exhibit a reduction of up to $37\times$ in the number of undetected flood requests compared to the alternative. Finally, we open-source the Memento algorithms and the HAProxy cloud load-balancer extension [3].

II. Why sliding windows?

We first show that sliding windows identify new heavy hitters quicker than interval methods. We consider accurate measurements, but the results are also valid for approximate ones.

Window vs. interval. We start by comparing *sliding windows* to the *Interval* method that is commonly used in HHH-based DDoS mitigation systems [41], [35], [37]. As depicted in Figure 1a, the Interval method relies on sequential interval measurements. Usually, the measurement data is available at the end of each measurement interval. We also consider an *improved* Interval method, in which it is accessible throughout each measurement period. There are two possible failure modes, namely: failing to detect a new heavy hitter (false negative) or falsely declaring a heavy hitter (false positive).

Ran Ben Basat is with the Department of Computer Science, University College London, London NW1 2AE, U.K.

Gil Einziger is with the Department of Computer Science, Ben Gurion University of the Negev, Beersheba 8410501, Israel.

Isaac Keslassy, Ariel Orda, and Shay Vargaftik are with the Department of Electrical and Computer Engineering, Technion–Israel Institute of Technology, Haifa 3200003, Israel.

Erez Waisbard is with Nokia Bell Labs, Kfar Sava 4439246, Israel.

This work was partly supported by the Israel Science Foundation (grant No. 1119/19), the Hasso Plattner Institute Research School, the Technion Hiroshi Fujiwara Cyber Security Research Center and the Israel Cyber Bureau, and by the Cyber Security Research Center at Ben-Gurion University.

Algorithms that follow the improved Interval method can have both false positives and false negatives. In contrast, sliding windows can avoid both errors. To show this, we start with the following definitions.

Definition 1 (Window Frequency)**.** *We denote by f_x^W the window frequency of flow x,* i.e., *the* number *of packets transmitted by x over the last W packets.*

Definition 2 (Normalized Window Frequency)**.** *We denote by $\frac{f_x^W}{W}$ the normalized window frequency of flow x,* i.e., *the* fraction *of x's packets within the last W packets.*

Next, window heavy hitters are flows whose normalized window frequency is larger than a user-defined threshold:

Definition 3 (Window Heavy Hitter)**.** *Flow x is a window heavy hitter if its normalized window frequency $\frac{f_x^W}{W}$ is larger than θ, where $\theta \in (0,1)$ is a user-defined threshold.*

We formalize the basic window frequency estimation problem:

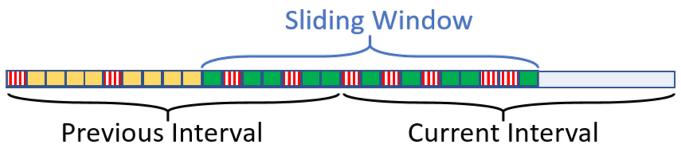
Definition 4. *An algorithm solves (ϵ, δ) -* WINDOW FREQUENCY ESTIMATION *if given a query for a flow (x), it provides $\widehat{f_x^W}$ such that* $\Pr\left[\left|f_x^W - \widehat{f_x^W}\right| \leq \varepsilon W\right] \geq 1 - \delta$.

Window optimality. The optimal detection point for new window heavy hitters is simply once their normalized window frequency is above a user-defined threshold. Reporting a flow earlier is *wrong* (false positive), and reporting it afterwards is *(too) late*. Thus, sliding window measurements, *by definition*, have an optimal detection time.

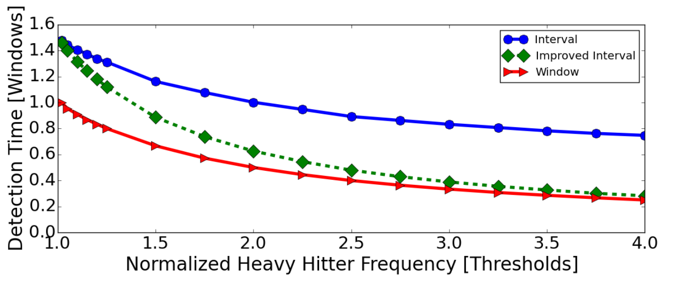
Motivation. We motivate the definition for window heavy hitters with an experimental scenario where a new flow appears during the measurement and consumes, at a constant rate, a larger-than-the-threshold portion of the traffic after its initial appearance. We measure how long it takes for each measurement method to identify the new heavy hitter and evaluate the following measurement methods:

(i) *Interval*: The window frequency of each flow is estimated at the end of every measurement. This method represents limitations of sampling techniques (*e.g.,* [21], [10]) that require time to converge and thus cannot provide estimates during the measurement. (ii) *Improved interval*: Same as *interval*, but flow frequencies are estimated upon the arrival of each packet. That is, Improved Interval finds heavy hitters once they appear enough times in the interval and does not need to wait to its end before reporting. This represents the best case scenario for the Interval method. (iii) *Window*: Sliding window, where frequencies are estimated upon packet arrivals.

Figure 1b plots the detection time for each method as a function of the normalized frequency of the new heavy hitter. Intuitively, larger heavy hitters are detected faster, because less time passes before their normalized window frequency reaches the threshold. Indeed, the *sliding window* approach is always faster than the *Interval* and *Improved Interval* methods. When the frequency is close to the detection threshold, we get up to 40% faster detection time compared to the Interval method. At the end of the tested range, sliding windows are still over 5% quicker. The Interval method is the slowest, as it estimates frequencies only at the end of the measurement. Thus, such a usage pattern is undesired for systems such as load balancing and attack mitigation.



(a) An example of the periodic *interval* and *sliding window* methods. In this scenario, consider a threshold of nine packets. The solid-green flow is a window heavy-hitter as it has ten packets within the sliding window. However, the measurement interval method does not detect the green flow, as it only has five packets within the current interval (false negative). Intuitively, one can identify the green flow by lowering the threshold to four packets, but in that case, the striped red flow is detected as well (false positive).



(b) Effect of a new heavy hitter's frequency on its detection time. The x-axis is the ratio of the normalized heavy hitter's frequency and the user-defined threshold. The y-axis is the expected detection time in windows. For instance, when the frequency is twice the threshold, it takes a window algorithm half a window to detect the new heavy hitter whereas interval-based algorithms require between 0.6-1.0 windows.

Fig. 1: Sliding windows compared to intervals.

III. SLIDING WINDOW ALGORITHMS

Our next step is to make sliding windows accessible to cloud operators. We do so by first introducing new single-device algorithms that are significantly faster than existing techniques, and then extend them to efficient network-wide algorithms that combine information from many measurement points to obtain a global perspective.

A. *Heavy Hitters on Sliding Windows*

Here, we present Memento – an algorithms that solves the (ϵ, δ) - WINDOW FREQUENCY ESTIMATION problem. First, we discuss a straw man approach.

Straightforward approach. Our goal is to produce faster sliding window algorithms. Intuitively, one can accelerate the performance of a heavy hitter algorithm by sub-sampling the packets. That is, we would like to sample packets with a probability of τ, use an HH algorithm with a window size of $W \cdot \tau$ packets, and then multiply its estimations by a factor of τ^{-1}. Unfortunately, this does not yield the desired outcome as the number of samples from the window varies, whereas sliding-window HH algorithms are designed for fixed-sized windows. Specifically, the actual number of samples in each W-sized window varies, which results in an additional error of $\pm\Theta\left(\sqrt{W(1-\tau)\cdot\tau^{-1}}\right)$ in the size

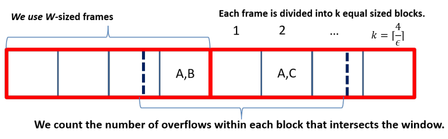

We use W-sized frames
Each frame is divided into k equal sized blocks.
1
2
...
$k = \lceil \frac{4}{\epsilon} \rceil$
A,B
A,C
We count the number of overflows within each block that intersects the window.

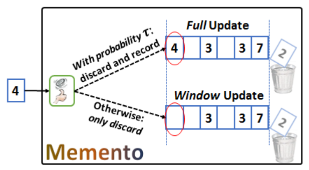
Fig. 2: High-level overview of the WCSS algorithm (adapted from a figure in [13]).

of the reference window. Since we are interested in small values of τ to achieve speedup (see Section IV-C), this approach results in a considerable error.

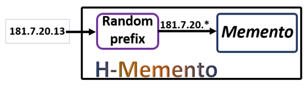
Memento overview. We observe that sliding window algorithms have two conceptual steps in the processing of packets to slide the window: discard the old information, and record the new one. The key idea behind *Memento* is to accelerate the computation by performing only the discarding step (which we term *Window Update*) for most packets, and additionally triggering the recording step (termed *Full Update*) with a small probability τ.

Therefore, Memento alternates between the fast Window updates and the slower Full updates. Full updates include (1) forgetting outdated data that is no longer in the window, *and* (2) adding a new item to the window. On the other hand, Window updates only involve maintaining the window. That is, Memento maintains a W-sized window but most of the packets within that window are missing. Thus, it attains speedup but avoids the additional error that is caused by uniform samples. The concept is exemplified in Figure 3a.

Implementation. For simplicity, we build Memento on top of an existing sliding window HH algorithm. This makes it easier to implement, verify, and then compare with the current approaches. We picked *Window Compact Space Saving (WCSS)* [13] as the underlying algorithm, but our approach is general and works on other window algorithms as well (*e.g.,* [12], [25]). Intuitively, when $\tau = 1$, Memento is identical to WCSS as it performs a full update for each packet. Figure 2 provides a brief overview of WCSS. WCSS divides the stream into W-sized frames and each frame into k equal-sized blocks. It uses a counting algorithm with a bounded error to count the appearances of each item within a frame. WCSS restarts the counting from zero once per frame, but this only affects the counts of non-heavy hitters. Each time heavy hitters reach a certain value, they 'overflow', and WCSS recalls the overflow events on a per-block base. The estimated value for each item depends on how many times it appears within blocks intersecting the window and on its estimation in the counting algorithm modulo the overflow value. By this description, observe that the update procedure of WCSS maintains the overflow list, forgets overflows that are guaranteed to be too old to affect the sliding window, and updates the counting algorithm. Also, notice that within the context of WCSS, the update procedure updates the illustrated window and forgets old overflows while also updating the approximate counting structure (providing an accuracy guarantee within a fixed amount of space). Thus, we can also see the natural partition to Full and Window updates.



(a) Memento, our HH algorithm, records the new item (4) with probability τ and always discards the old data (2). Speedup is achieved by rarely performing Full updates.



(b) H-Memento, our HHH algorithm, simply updates Memento with a single random prefix, achieving constant time complexity.

Fig. 3: High-level overview of our algorithms.

Algorithm 1 describes Memento (note that in the special case of $\tau = 1$ Memento coincides with WCSS [13]). We'll eventually use a union bound to bound the total error of Memento (ϵ) as the sum of these two errors. The error in Memento comes from the fact that the underlying heavy hitter algorithm provides approximate frequencies (ϵ_a) and from the fact that we only updated it for sampled packets (ϵ_s). So Memento exposes a tradeoff, we can reduce ϵ_a to increase ϵ_s the former results in higher memory consumption, while the latter in faster operation. Thus we trade some memory efficiency for speed.

As detailed in Algorithm 1, given an algorithmic parameter ϵ_a such that $W \cdot \epsilon_a \gg 1$, Memento divides the stream into *frames* of size W, where each frame is then further partitioned into $k \triangleq \left\lceil \frac{4}{\epsilon} \right\rceil$ equal-sized *blocks.*

Memento follows on similar lines as WCSS [13] but we separate WCSS's update process into a *Window* update that updates the sliding window and ages the counters, and a *Full* update that reports a new packet. For completeness, we now describe the update process of Memento. Memento uses a regular heavy hitter algorithm (e.g., Space Saving [30]) to count how many times each item arrives within the last *frame*, where a frame is a W fixed W sized segment. E.g., each sliding window intersects either a single frame if the packet count is a multiplicative of W, and two frames otherwise. Heavy hitter algorithms conserve space and provide us with a bounded estimation error. Since we have multiple sources of error, we allocate that algorithm with $k = \lceil 4/\epsilon_a \rceil$ which yields an estimation bound of at most $W/k \leq W\epsilon/4$ for the number of occurrences within the current frame. Whenever the estimation of an item reaches a multiple of ($W/k \leq W\epsilon/4$)-a size which we call a *Block* we record such an event as *overflow*, which we will remember for W packets.

At the end of the frame, we reset the heavy hitter instance, which adds at most $W/k \leq W\epsilon/4$ to the estimation error of all flows. This action implies that we only remember the overflows within a frame and forget the rest. Still, it allows us to reuse the heavy hitter algorithm for the new frame.

We now explain how Memento remembers how many times each flow overflowed within a frame for exactly W packets since the overflow events. Memento using a queue of queues (b) that contains one queue for each block that overlaps with the current window. That is, we use $k+1$ queues in total. Each queue in b contains an ordered list of items that overflowed in the corresponding block. When a block ends, we remove the oldest queue from b, as it no longer overlaps with the window. Additionally, we append a new empty queue to b. Note that Memento does not count accurately but instead uses a Space Saving [30] instance (denoted y) to approximately count the in-frame frequency. Critically, when adding up to W elements, we are guaranteed that any flow with a size of at least W/k will not lose its counter (as the sum of all counters is at most W, the minimum cannot be larger than W/k). Roughly speaking, Space Saving uses k counters to track the frequencies of all flows. Whenever a flow without a counter arrives, the minimal counter is disassociated with its flow and allocated for the current one (but is incremented as if it was the current flow's counter). This property ensures that once there is an overflow (Line 15) x does lose its counter until the end of the frame and is accurately tracked from now on. Finally, we clear y at the end of each frame.

The frequency of an item is estimated by multiplying its number of overflows by the block size and adding the remainder of its appearance count as reported by y. In [33], Mitzenmacher, Steinke and Thaler (MST) suggest an HHH detection algorithm with proven approximation guarantees and a one-sided error (the algorithm never under-approximates). Thus, we choose to keep the error one-sided as well for comparison purposes. To do so, Memento adds $2\frac{W}{k}$ to each item's estimation. It then multiplies the result by τ^{-1} as a Full update is performed on average once per τ^{-1} packets. The table B counts the number of overflows for each item for quick frequency queries. Memento de-amortizes the update of $B[x]$, achieving constant worst case time. To that end, when processing a packet, Memento pops (at most) one flow from the queue of the oldest block (see lines 8-11). This ensures that the worst case update time is constant as we are guaranteed that by the end of the block we will have an empty queue and B will be fully updated. Finally, for finding the heavy hitters themselves (rather than just estimating flow sizes), Memento iterates over the flows with entry in B and estimate their sizes. Since every heavy hitter must overflow in the window, we are guaranteed that it will have such an entry.

Algorithm 1 Memento (W, ϵ, τ)

```
1: Initialization: $k = \lceil 4/\epsilon \rceil$, $y = SpaceSaving.init(k)$, $M = 0$, $B =$ Empty hash, $b =$ Queue of $k+1$ empty queues.
2: function WindowUpdate()                                    ▷ discard old information
3:     M = M + 1 $\mod W$
4:     if $M = 0$ then $y$.FLUSH()                                   ▷ new frame
5:     if $M \mod \frac{W}{k} = 0$ then                               ▷ new block
6:         $b$.POP()                ▷ Remove *one* item from the queue, if such exists
7:         $b$.APPEND(new empty queue)
8:     if $b$.tail is not empty then                         ▷ remove oldest item
9:         $oldID = b$.tail.POP()    ▷ Remove the oldest *queue* as its block left the window
10:        $B[oldID] = B[oldID] - 1$
11:        if $B[oldID] = 0$ then $B$.REMOVE($oldID$)   ▷ $oldID$ has no more overflows, delete to save space
12: function FullUpdate(Item x)                 ▷ add item and discard old information
13:     WindowUpdate()
14:     $y$.ADD($x$)                                   ▷ add item $x$ to the Space Saving
15:     if $y$.QUERY($x$) $mod \frac{W}{k}$ = 0 then ▷ $x$ has overflowed, add it to the queue of the block
16:         $b$.head.PUSH($x$)
17:         if $B$.CONTAINS(x) then $B[x] = B[x] + 1$
18:         else $B[x] = 1$                                            ▷ adding $x$ to $B$
19: function Update(Item x)
20:     if $Uniform(0,1) \leq \tau$ then FullUpdate($x$)              ▷ With probability $\tau$
21:     else WindowUpdate()
22: function Query(Item x)                  ▷ Estimate the window-frequency of item $x$
23:     if $B$.Contains($x$) then                       ▷ If $x$ had at least one overflow
24:         return $\tau^{-1} \cdot \left(\frac{W}{k} \cdot (B[x] + 2) + \left(y.\text{QUERY}(x) \mod \frac{W}{k}\right)\right)$
25:     else return $\tau^{-1} \cdot \left(2\frac{W}{k} + y.\text{QUERY}(x)\right)$                    ▷ no overflows
```

B. Extending to Hierarchical Heavy Hitters

HHHs monitor subnets and flow aggregates in addition to individual flows. We start by introducing existing approaches for HHH measurements on sliding windows.

Existing approaches. In MST [33], multiple HH instances are used to solve the HHH problem. MST monitors each prefix length separately with a dedicated Space Saving and extracts the HHH prefixes by combining the data from all instances. This design trivially extends to sliding windows by replacing the HH building blocks with window algorithms (*e.g.,* WCSS [13]). This was proposed by [33] but dismissed as impractical. Replacing the underlying algorithms with Memento is slightly better as we can perform Window updates to most instances. Unfortunately, the update complexity remains $\Omega(H)$, where H is the size of the hierarchy (typically between 5 to 1089), as we explain below. For many HHH flavors, the update complexity is too slow for real time analysis. In contrast, H-Memento achieves constant time updates, matching the complexity of interval algorithms [10]. Another intuitive approach comes from the *Randomized Heirarchical Heavy hitters (RHHH)* [10] algorithm. RHHH shares the same data structure as MST but randomly updates at most a single HH instance which allows for constant time updates. Specifically, for each packet RHHH chooses a single random prefix and updates it at the relevant HH instance. Intuitively, if a prefix is sufficiently frequent, it will be sampled enough times to be identified. Additionally, it makes small changes to the query procedure to compensate for the sampling error and guarantees that (with high probability) it will have no false negatives. This method does not work for sliding windows, as each HH instance is updated a varying number of times and monitors a possibly different window.

H-Memento's overview. In H-Memento we maintain a single large Memento instance and use it to monitor all the sampled prefixes which is the main difference from previous approaches [10], [33] that use a separate approximate counting algorithm for each prefix type. Our structure means that we maintain a single sliding window to measure all subnets, which the underlying Memento does in constant time. This approach also has engineering benefits such as code reuse, simplicity, and maintainability. The update procedure of H-Memento is illustrated in Figure 3b. Next, we proceed with notations and definitions for the HHH problem, which we later use to detail H-Memento.

HHH notations and definitions. For brevity, Table I summarizes the notations used in this work. These definitions extend the problem formulation [10] to sliding windows. We consider IP prefixes (*e.g.,* 181.$*$). A prefix without a wildcard (*e.g.,* 181.7.20.6) is called *fully specified.* The notation $\mathcal{U}$ is the domain of the fully specified items. For example, $\mathcal{U}$ can be all the IPv4 (or IPv6) source addresses, all pairs of such addresses, all

Symbol	Meaning	Symbol	Meaning
$\mathbb{S}$	The packet stream.	N	Current number of packets (the stream length).
W	The window size.	H	Size of the hierarchy.
τ	Sampling probability.	V	Sampling ratio for HHH, $V \triangleq \frac{H}{\tau}$.
S_x^i	Variable for the i'th appearance of a prefix x.	S_x	Sampled prefixes with id x.
S	Sampled prefixes from all ids.	$\mathcal{U}$	Domain of fully specified items.
$\epsilon, \epsilon_s, \epsilon_a$	Overall, sample, algorithm's error guarantee.	$\delta, \delta_s, \delta_a$	Overall, sample, algorithm's confidence.
θ	Threshold parameter.	$C_{q\|P}$	Conditioned frequency of q with respect to P.
$G(q\|P)$	Subset of P with the closest prefixes to q.	f_q^W	The window frequency of prefix q.
$\widehat{f_q^W}^+, \widehat{f_q^W}^-$	Upper and lower bounds for f_q^W.	Z	inverse CDF of the normal distribution.
$\mathcal{B}$	per-packet control bandwidth budget.	$\mathbb{O}$	the minimal header size (in bytes),
E	bytes required to report a packet.	m	number of measurement points.
b	number of samples in each report.	$\mathfrak{E}_b$	overall error in network-wide settings.
L	Depth of the hierarchy.	H_p and H_P	The fully specified elements generalized by prefix p and prefix set P.

TABLE I: Summary of notations.

5-tuples, etc. A prefix p_1 *generalizes* another prefix p_2 if p_1 is a prefix of p_2. For example, 181.7.20.$*$ and 181.7.$*$ generalize the (fully specified) 181.7.20.6. The *parent* of a prefix is the longest generalizing prefix, *e.g.,* 181.7.$*$ is 181.7.20.$*$'s parent. The generalization does not have to be at byte-level granularity as in the above example. Our approach is also applicable to bit-level hierarchies; for example, using IPv4 source address bit-level we have $208.67.222.222/31 \preceq 208.67.222.220/30$.

The *size of the hierarchy (H)* is the number of different prefixes that generalize a fully specified prefix. For example, in byte granularity one dimensional heavy hitters. The fully specified prefix 123.234.68.1 is generalized by itself, by 123.234.68.$*$, by 123.234.$*$.$*$, by 123.$*$.$*$.$*$, and by $*$ and thus $H = 5$. Similarly, $H = 33$ for bit-level hierarchy.

Hierarchies can also be multi-dimensional. For example, we can consider tuples of the form (source IP, destination IP). In that case, fully specified "prefixes" are fully determined in both dimensions, *e.g.,* $(\langle 181.7.20.6\rangle, \langle 208.67.222.222\rangle)$. Also, observe that "prefixes" now have two parents, *e.g.,* $(\langle 181.7.20.*\rangle, \langle 208.67.222.222\rangle)$ and $(\langle 181.7.20.6\rangle, \langle 208.67.222.*\rangle)$ are both parents to $(\langle 181.7.20.6\rangle, \langle 208.67.222.222\rangle)$.

Definition 5 formalizes the generalization concept.

Definition 5 (Generalization)**.** *Let p, q be prefixes. We say that p generalizes q and denote $q \preceq p$ if for each dimension i, $p_i = q_i$ or $q_i \preceq p_i$. We denote the set of fully specified items generalized by p using $H_p \triangleq \{e \in \mathcal{U} \mid e \preceq p\}$. Similarly, the set of every fully specified item that is generalized by a set of prefixes P is denoted by: $H_P \triangleq \cup_{p\in P} H_p$. Moreover, denote $p \prec q$ if $p \preceq q$ and $p \neq q$.*

Definition 5 also deals with the multidimensional case.

Next, we consider a set of prefixes P and denote $G(p|P)$ as the set of prefixes in P that are most closely generalized by the prefix p. That is, $G(p|P) \triangleq \{h : h \in P, h \prec p, \nexists\, h' \in P \; s.t. \; h \prec h' \prec p\}$.

For example, consider the prefix $p =< 142.14.* >$ and the set $P = \{< 142.14.13.* >, < 142.14.13.14 >\}$, then we have $G(p|P) = \{< 142.14.13.* >\}$. The window frequency of a prefix p is the total sum of packets within the window that are generalized by p, *i.e.,* $f_p^W \triangleq \sum_{e\in H_p} f_e^W$. Note that each packet is generalized by H different prefixes. This motivates us to look at the *conditioned* (residual) frequency that a prefix p adds to a set of already selected prefixes P. The conditioned frequency is defined as: $C_{p|P} \triangleq \sum_{e\in H_{P\cup\{p\}}\setminus H_P} f_e^W$.

We denote by X_p^W the number of times prefix p is sampled in the window, $\widehat{X_p^W}^+$ is an upper bound on X_p^W and $\widehat{X_p^W}^-$ is a lower bound. The notation $V \triangleq \frac{H}{\tau}$ stands for the sampling rate of each specific prefix. We define:
$\widehat{f_p^W} \triangleq \widehat{X_p^W} V$ – an estimator for p's frequency.
$\widehat{f_p^W}^+ \triangleq \widehat{X_p^W}^+ V$ – an upper bound for p's frequency.
$\widehat{f_p^W}^- \triangleq \widehat{X_p^W}^- V$ – a lower bound for p's frequency.

We now define the *depth* of a prefix (or a prefix tuple). Fully specified items are of depth 0, their parents are of depth 1 and more generally, the parent of an item with depth x is of depth $x+1$. L denotes the maximal depth; In a single dimension, $L = H$, but in multiple dimensions L is smaller than H because each prefix has two parents. Hierarchical heavy hitters are calculated by iterating over all fully specified items (depth 0). If their frequency is larger than a threshold of θW, we add them to the set HHH_0. Then, we go over all the items with depth 1 and if their *conditioned* frequency, with regard to HHH_0, is above θW, we **add** them to the set. We denote the resulting set as HHH_1 and repeat the process L times until the set HHH_L contains the (exact) hierarchical heavy hitters. Unfortunately, we need space that is linear in the stream size to calculate exact HHH (and even plain heavy hitters) [36]. Hence, as done by previous work [10], [33], [18], [19], [20], we solve *approximate HHH*.

More specifically, a solution to the approximate HHH problem is a set of prefixes that satisfies the *Accuracy* and *Coverage* conditions (Definition 6). Here, "Accuracy" means that the estimated frequency of each prefix is within acceptable error bounds and "Coverage" means that the conditioned frequency of prefixes not included in the set is below the threshold. This does not mean that the conditioned frequency of prefixes that are included in the set is above the threshold. Thus, the set may contain a small number of subnets misidentified as HHH (false positives).

Definition 6 (Approximate HHHs)**.** *An algorithm $\mathbb{A}$ solves $(\delta, \epsilon, \theta)$ -* Approximate Window Hierarchical Heavy

Algorithm 2 H-Memento (W, ε_a, τ)

```
Initialization: $Memento.init(H \cdot \epsilon_a^{-1}, W, \tau \cdot H)$
1: function UPDATE( $x$)
2:     $Memento.update(RandomPrefix(x))$
3: function OUTPUT($\theta$)
4:     $HHH = \phi$
5:     for Level $\ell = 0$ up to $L$ do
6:         for each $p$ in level $\ell$ do          ▷ Only over prefixes with a counter.
7:             $\widehat{C_{p|HHH}} = \widehat{f_p^W}^+ + calcPred(p, HHH)$
8:             $\widehat{C_{p|HHH}} = \widehat{C_{p|HHH}} + 2Z_{1-\delta}\sqrt{VW}$ ▷ Compensate for sampling
9:             if $\widehat{C_{p|HHH}} \geq \theta N$ then $HHH = HHH \cup \{p\}$
10:    return $HHH$
```

Algorithm 3 calcPred for one dimension

```
1: function CALCPRED(prefix $p$, set $P$)
2:     $R = 0$
3:     for each $h \in G(p|P)$ do $R = R - \widehat{f_h^W}^-$
4:     return $R$
```

HITTERS *if for any run of the algorithm it returns a set of prefixes P that satisfies:*

Accuracy: *If* $p \in P$ *then* $\Pr\left(\left|f_p^W - \widehat{f_p^W}\right| \leq \varepsilon W\right) \geq 1 - \delta$.

Coverage: *If* $q \notin P$ *then* $\Pr\left(C_{q|P} < \theta W\right) \geq 1 - \delta$.

H-Memento's full description. A pseudo-code for H-Memento is given in Algorithm 2. The output method performs the HHH set calculation as explained for exact HHH. The calculation yields an approximate result as we only have an approximation for the frequency of each prefix. Thus, we conservatively estimate conditioned frequencies.

For two dimensions, we use inclusion-exclusion (Definition 7, herein) to avoid double counting.

Definition 7. *Given prefixes h, h', the greatest lower bound of h, h' is the most generalized shared descendant of h, h'. That is, denoted by $q = glb(h, h')$, the greatest lower bound satisfies: $\forall p : ((p \preceq h) \wedge (p \preceq h') \Rightarrow p \preceq q)$. If h and h' have no common descendants, $glb(h, h') = 0$.*

Pseudo-code for the update method is given in Algorithm 2, which is the same for one and two dimensions. The difference between these is encapsulated in the CALCPRED method which uses Algorithm 3 for one dimension and Algorithm 4 for two. In two dimensions, $C_{p|HHH}$ is first set in Line 7 of Algorithm 2. Then, we remove previously selected descendant heavy hitters (Line 3, Algorithm 4) and finally we add back the common descendants (Line 6, Algorithm 4). The sampling error is accounted for in Line 8. Intuitively, our analysis shows which τ values guarantee that H-Memento solves the approximate HHHs problem. A formal proof of the algorithm's correctness appears in Section VI.

C. Network-Wide Measurements

As Figure 4 illustrates, we now discuss a centralized controller that receives data from multiple clients and forms a network-wide view of the traffic (*e.g.,* network-wide HH or HHH). Similarly to [22], [6] we assume that there are several measurement points and that each packet is measured once. Our design focus is on two critical aspects of this system: (1) a *communication method* between the clients and the controller that conveys the information in a timely and bandwidth-efficient manner, and (2) a fast *controller algorithm.*

Algorithm 4 calcPred for two dimensions

```
1: function CALCPRED(prefix $p$, set $P$)
2:     $R = 0$
3:     for each $h \in G(p|P)$ do $R = R - \widehat{f_h^W}^-$
4:     for each pair $h, h' \in G(p|P)$ do
5:         $q = glb(h, h')$
6:         if $\nexists h_3 \neq h, h' \in G(p|P), q \preceq h_3$ then $R = R + \widehat{f_h^W}^+$
7:     return $R$
```

Formal model. First, we define a sliding window in the network-wide model as the last W packets that were measured somewhere in the network. Intuitively, we want the controller to analyze the traffic of the most recent W packets *in the entire network*, as monitored by the measurement points. When considering natural sampling based communication methods, the main challenge is to maintain the sliding window despite not knowing the exact order, and the number of packets. We extend Memento's sampling approach, as well as the natural communication patterns to allow for provably accurate measurement with these methods. Specifically, whenever a measurement point reports to the controller, it includes the sampled packets as well as the total number of unsampled packets. In turn, Memento performs Window updates for each unsampled packet and Full updates for each sampled packet. This approach maintains the measurement with respect to a fixed sized window, but we remain with uncertainty about the correct packet order and may include outdated packets in the window. However, we formally establish that this error is controllably small.

(1) Communication method. We now propose three controller algorithms for common communication patterns. Each method must adhere to a per-packet bandwidth budget ($\mathcal{B}$). This implies that smaller reports can be sent more frequently but also deliver less information.

Aggregation. The content of two HH instances can often be efficiently merged [8]. MST [33] and RHHH [10] use HH algorithms as building blocks, and can, therefore, be merged as well. This capability motivates the *Aggregation* communication method where each client periodically transmits all the entries of its HH instances to the controller. Given enough bandwidth, this method is intuitively the most communication-efficient, as all data is transmitted. However, as each message is large, we infrequently report to the controller to meet the bandwidth budget.

Sample. Most network devices are capable of transmitting uniform packet samples to the controller. Motivated by this capability, the Sample method samples packets with a fixed probability τ, and sends a report to the controller once per τ^{-1} packets. Thus on average, each message contains a single sample. This information is enveloped by the usual packet headers that are required to deliver the packet in the network. We observe that this uses a significant portion of the bandwidth for the header fields of the transmitted packet. Yet, the Sample method is considerably easier to deploy than the Aggregate option, as the nodes only sample packets and do not run the measurement algorithms. The communication pattern is

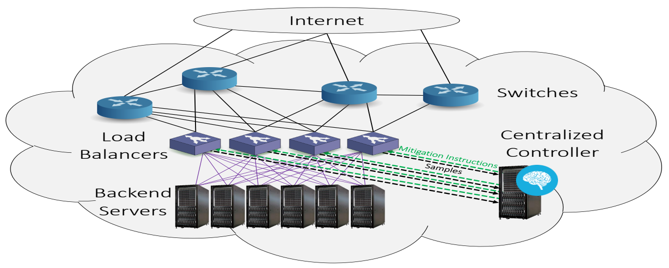


Fig. 4: An overview of our system. The clients (load-balancers) perform the measurements and periodically send information to a centralized controller. The controller then runs a global sliding-window analysis. For example, in the case of an HTTP flood, it can mitigate the attack by instructing the clients which subnets to rate-limit or block.

network-friendly as we get a stable flow of traffic from the clients to the controller.

Batch. The Batch approach balances between reporting delay and bandwidth efficiency. The idea is simple: we send on average b samples (*e.g.,* 100) per report. That is, we send a report once per $\tau^{-1}b$ packets, containing all the sampled packets within this period. Our analysis finds the the optimal batch size b that minimizes the total error.

(2) Controller algorithm. The controller maintains an instance of Memento or H-Memento where we term the respective algorithms D-Memento and D-H-Memento. The controller behaves slightly differently in each option.

Aggregation. Aggregation is used in this study only as a baseline. Thus, instead of implementing a specific algorithm, we simulate an *idealized* aggregation technique with an unlimited space at the controller and no accuracy losses upon merging.

Sample and Batch. In the Sample and Batch schemes, the controller maintains a Memento or H-Memento instance. When receiving a report, it first performs a Full update for each sampled packet and then makes Window updates for the un-sampled ones. In total, the Sample performs τ^{-1} updates and the Batch performs $\tau^{-1}b$ updates.

IV. Evaluation

Server. Our evaluation was performed on a Dell 730 server running Ubuntu 16.04.01 release. The server has 128GB of RAM and an Intel Xeon CPU E5-2667 v4@ 3.20GHz.

Traces. We use real packet traces collected from an edge router *(Edge)* [5], a datacenter *(Datacenter)* [15], and a CAIDA backbone link *(Backbone)* [23].

Algorithms and implementation. For the HH problem, we compare Memento and WCSS [13]. For WCSS we use our Memento implementation without sampling ($\tau = 1$). For the HHH problem, we compare H-Memento to MST [33] and RHHH [10] (interval algorithms). We use the code released by the original authors of these algorithms. We also form the *Baseline* sliding window algorithm by replacing the underlying algorithm in MST [33] with WCSS. Specifically, MST proposed to use Lee and Ting's algorithm [28] as WCSS was not known at the time. By replacing the algorithm with the WCSS, a state of the art window algorithm, we compare with the best variant known today.

Yardsticks. We consider source IP hierarchies in byte granularity ($H = 5$) and two-dimensional source/destination hierarchies ($H = 25$). Such hierarchies are also used in [10], [33], [20]. We run each data point 5 times and use two-sided Student's t-tests to determine the 95% confidence intervals. We evaluate the empirical error in the *On Arrival model* [13], [14], where for each packet we estimate its flow (denoted s_t) size. We then calculate the *Root Mean Square Error (RMSE)*, *i.e.,* $RMSE(Alg) \triangleq \sqrt{\frac{1}{|N|}\sum_{t=1}^{N}(\widehat{f_{s_t}^W} - f_{s_t}^W)^2}$.

A. Heavy Hitters Evaluation

We evaluate the effect of the sampling probability τ on the operation speed and empirical accuracy of Memento, and use the speed and accuracy of WCSS as a reference point for this evaluation. The notation X-WCSS stands for WCSS that is allocated X counters (for X$\in \{64, 512, 4096\}$). Similarly, X-Memento is for Memento with X counters. The window size is set to $W \triangleq 5$ million packets and the interval length is set to $N \triangleq 16$ million packets.

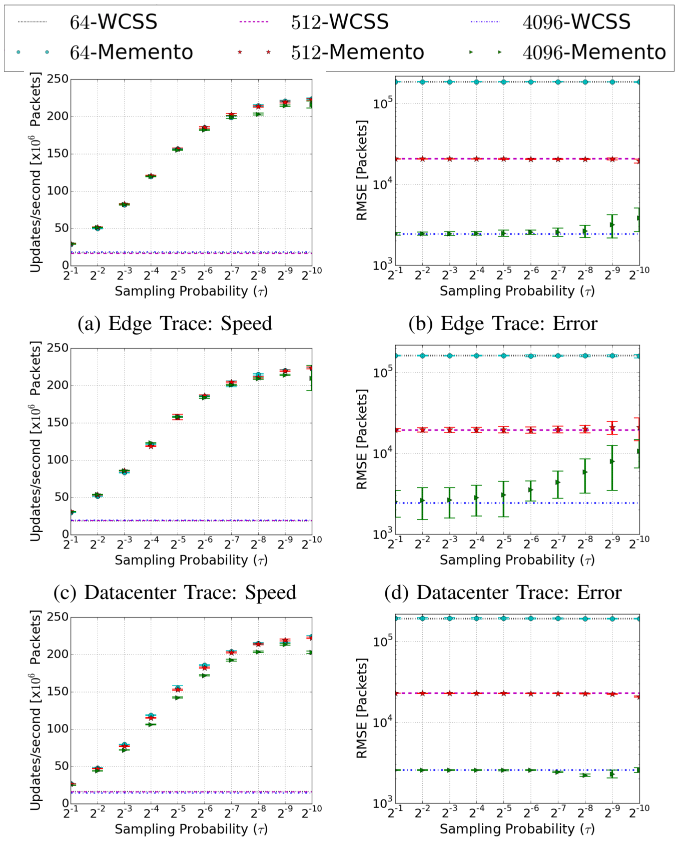
As depicted in Figure 5, the update speed is determined by the sampling probability and is almost indifferent to changes in the number of counters. Memento achieves a speedup of up to 14$\times$ compared to WCSS. As expected, allocating more counters also improves the accuracy. It is also evident that the error of Memento is almost identical to that of WCSS, which indicates that it works well for the range. The smallest evaluated τ, namely, $\tau = 2^{-10}$, already exhibits slight accuracy degradation, which shows the limit of our approach. It appears that a larger number of counters, or heavy-tailed workloads (such as the Backbone trace), allow for even smaller sampling probabilities without a impact to the attained accuracy.

B. Hierarchical Heavy Hitters Evaluation

H-Memento vs. existing window algorithm. Next, we evaluate H-Memento and compare it to the Baseline algorithm. We consider two common types of hierarchies, namely a one-dimensional source hierarchy ($H = 5$) and two-dimensional source/destination hierarchies ($H = 25$). Note that H-Memento performs updates in constant time while the Baseline does it in $O(H)$. Following the insights of Figure 5, we evaluate H-Memento with a sampling rate τ such that $\tau \geq H \cdot 2^{-10}$, so that each of the H prefixes is sampled with a probability of at least 2^{-10}. That is, we do not allow sampling probabilities of $\tau < H \cdot 2^{-10}$ to get an effective sampling rate of at least 2^{-10}, which is the range in which Memento is accurate.

We evaluate three configurations for each algorithm, with a varying number of counters. The notation 64H denotes the use of $64 \cdot 5 = 320$ counters when $H = 5$, and 1600 counters when $H = 25$. The notations 512H and 4096H follow the same rule. In the Baseline algorithm, the counters are utilized in H equally-sized WCSS instances, while H-Memento has a single Memento instance with that many counters.

Figure 6 shows the evaluation results. We can see how τ is the dominating performance parameter. H-Memento achieves

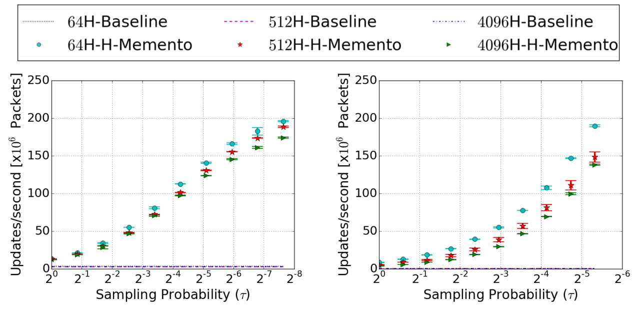


(a) Edge Trace: Speed (b) Edge Trace: Error

(c) Datacenter Trace: Speed (d) Datacenter Trace: Error

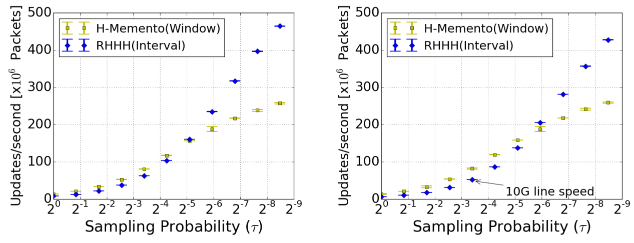
(e) Backbone Trace: Speed (f) Backbone Trace: Error

Fig. 5: Effect of the sampling probability τ on the speed and accuracy for a given memory constraint (*i.e.,* number of counters) given three different traces and a window size $W = 5M$. Memento is considerably faster than WCSS, but the accuracy of both algorithms is almost the same despite Memento's use of sampling, except when the sampling rate is low and the number of counters is high. Even then, this is mainly evident in the skewed Datacenter trace.



(a) Backbone trace - 1 D (H=5) (b) Backbone trace - 2 D (H=25)

Fig. 6: Effect of the sampling probability on the speed of $H\text{-}Memento$, compared to the Baseline algorithm in the Backbone trace. Note that H-Memento achieves a speedup of up to 53× in 1D and up to 273× in 2D. Results for the Edge and Datacenter traces are similar.



(a) One dimension (H=5) (b) Two dimensions (H=25)

Fig. 7: Speed comparison between RHHH (interval algorithm) and H-Memento (window algorithm) on the Backbone dataset. The annotated point shows the throughput of the $(\tau = 1/10)$-RHHH algorithm that is reported to meet the 10G line speed using a single core [10]. That is, H-Memento is slightly faster than RHHH in the parameter range of 10G lines.

up to a 52× speedup in source hierarchies and a 273× speedup in source/destination hierarchies. This difference is explained by the fact that the Baseline algorithm makes H expensive Full updates for each packet, while H-Memento usually performs a single Window update.

H-Memento vs. interval algorithm. Next, we compare the throughput of H-Memento to the previously suggested RHHH [10]. H-Memento and RHHH are similar in their use of samples to increase performance. Moreover, RHHH is the fastest known interval algorithm for the HHH problem. Our results, presented in Figure 7, show that $H\text{-}Memento$ is faster than RHHH for small sampling ratios. The reason lies in the implementation of the sampling. Namely, in RHHH, sampling is implemented as a geometric random variable, which is inefficient for small sampling probabilities, whereas in $H\text{-}Memento$, it is performed using a random number table. To see this, notice that the heavy operation in geometric variation is the coin-toss that happens once for every sampled packet. For unsampled packets, one just needs to decrement a counter until the next sample. Using a random number table, we avoid the coin-toss and only increment the offset in the table. Notice that this is a tradeoff, as the table requires additional space.

Still, as the sampling probability gets lower, the geometric calculation becomes more efficient, and eventually, RHHH is faster than $H\text{-}Memento$. This is because H-Memento performs a Window update for most packets, while RHHH only decrements a counter. Looking at both performance figures independently, we conclude that $H\text{-}Memento$ achieves very high performance and is likely to incur little overheads in a virtual switch implementation in a similar manner to RHHH.

C. Network-Wide Evaluation

This section describes our proof-of-concept system. We incorporated H-Memento into HAProxy which provides the capability to monitor traffic from subnets which allows to rate limit subnets. Our controller periodically receives information from (in the Batch, Sample or Aggregate method) the load-balancers and uses it to perform the HHH measurement (with the D-H-Memento algorithm). Then, the HHH output is used as a threshold-based attack mitigation where a subnet is rate-limited if its window frequency is above the threshold.

HAProxy. We implemented and integrated our algorithms into the open-source HAProxy load-balancer (version 1.8.1). Specifically, we leveraged and extended HAProxy's Access

Control List (ACL) capabilities, to allow the updates of our algorithms with new arriving data as well as to perform mitigation (*i.e.,* Deny or Tarpit) when an attacker is identified.

Traffic generation. Our goal is to obtain realistic measurements involving multiple simultaneous stateful connections such as HTTP GET and POST requests from multiple clients towards the load-balancers. To that end, we developed a tool that enables a single commodity desktop to maintain and initiate stateful HTTP GET and POST requests sourcing from multiple IP addresses. Our solution requires the cooperation of both ends (*i.e.,* the traffic generators and the load-balancer servers) for an arbitrarily large IP pool.

It is based on the *NFQUEUE* and *libnetfilter*-queue Linux targets that enable the delegation of the decision on packets to a userspace software. As reported by the Apache *ab load* testing tool, using a single commodity computer, we can initiate and maintain up to 30,000 stateful HTTP requests per second from different IPs without using the HTTP keep-alive feature. We are only limited by the pace at which the Linux kernel opens and closes sockets (*e.g.,* TCP timeout).

Controller. We implemented in C a test controller that communicates with the load-balancers via sockets. It holds a local HHH algorithm implementation and exchanges information with the load-balancers (*e.g.,* receives aggregations, samples, or batches). The controller then generates a global and coherent window view of the ingress traffic.

Testbed. We built a testbed involving three physical servers. The first is used for traffic generation towards the load-balancers. Specifically, we used several apache ab instances augmented with our tool to generate realistic stateful traffic from multiple IP addresses with delay and racing among different clients. The second station holds ten autonomous instances (*i.e.,* separate processes) of HAProxy load-balancers listening on different ports for incoming requests. Finally, at the third station, we used docker to deploy Apache server instances listening on different sockets.

1) H-Memento's Accuracy: In this experiment, we evaluate MST (denoted as *Interval*), the Baseline algorithm and H-Memento with a single load-balancer client. Our goal is to monitor the last 1,000,000 HTTP requests that have entered the load-balancer. The Baseline algorithm and H-Memento are set at $\epsilon_a = 0.1\%$ and a window size of 1,000,000 requests. The MST Interval instance is using a measurement period of 1,000,000 requests and is configured with $\epsilon_a = 0.025\%$, resulting in comparable memory usage. For each new incoming HTTP request, each algorithm estimates the frequency of each of its IP prefixes. The results are depicted in Figure 8. In all the traces, the Interval approach is the least accurate, while as expected, H-Memento is slightly less accurate than the Baseline algorithm due to its use of sampling. These conclusions hold for every prefix length and testbed workload.

2) Accuracy and Traffic Budget: In this experiment, we generate traffic towards ten load-balancers communicating with a centralized controller that maintains a global window view of the last 1,000,000 requests that entered the system. We evaluate the three different transmission methods (Aggregation, Sample, and Batch) with the same 1-byte per packet control traffic budget.

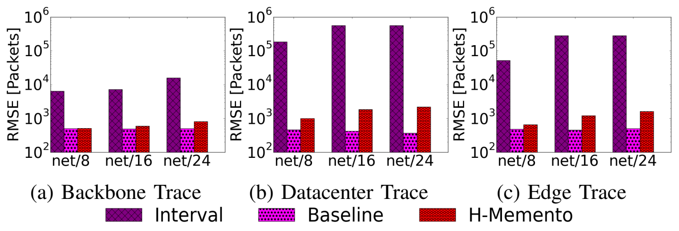


Fig. 8: Comparing the error of H-Memento.

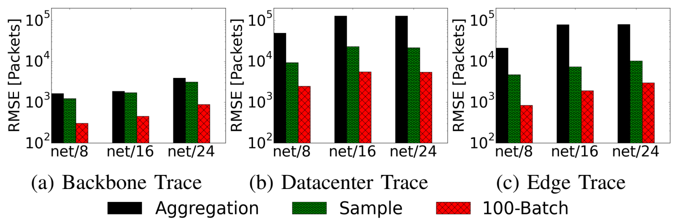


Fig. 9: Network-wide evaluation. Accuracy attained by D-H-Memento with a bandwidth limit of 1B per ingress packet under different transmission options.

Results. Figure 9 depicts the results. As indicated, the best accuracy is achieved by the Batch approach, while Sample significantly outperforms Aggregation. Intuitively, the Aggregation method sends the largest messages, each of which contains the full information known to the measurement point. Its drawback is a long delay between controller updates. The Sample method has a smaller delay but utilizes the bandwidth inefficiently due to the packet header overheads. Finally, Batch has a slightly higher delay but delivers more data within the bandwidth budget, which improves the controller's accuracy.

D. HTTP Flood Evaluation

Our deployment consists of ten HAProxy load-balancers that serve as the entry point and direct requests to Apache servers. The load-balancers report to the centralized controller that discovers subnets that exceed the user-defined threshold. The bandwidth budget is set to 1-byte per packet and the window size is $W = 1$ million packets.

Traffic. We inject flood traffic on top of the Backbone packet trace. Specifically, we select a random time at which we inject 50 randomly-picked 8-bit subnets that account for 70% of the total traffic once the flood begins. We generate a new trace as follows. (1) We select 50 subnets by randomly choosing 8-bits for each, and (2) a random trace line in the range $(0,10^6)$. Until that line the trace is unmodified. (3) From that line on, with probability 0.7 we add a flood line from a uniformly picked flooding sub-network.

This attack is motivated by recent efforts (e.g., [2], [1], [4]) in protecting against HTTP floods generated by swarms of infected devices via botnets. Such attacks aim to generate high volume traffic from many different geographic locations.

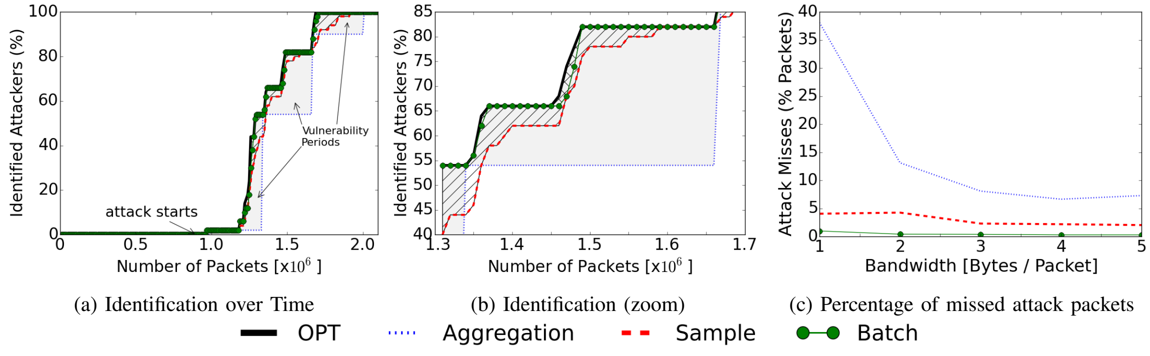


Fig. 10: Detection time in HTTP flood detection experiment with 50 attacking LANs on top of the backbone trace.

Results. Figure 10 depicts the results. Figure 10a and Figure 10b show the detection speed of the flooding subnets by the three different approaches at the controller. We compare among the three approaches and additionally outline an optimal algorithm that uses an accurate window and "knows" exactly what traffic enters the load-balancers without delay (OPT). It is notable that the Batch approach achieves near-optimal performance, and outperforms Sample and Aggregation. Figure 10c shows that the Batch method identifies almost all of the attack messages as is expected by our theoretical analysis. Further, its miss rate is 37$\times$ smaller under the 1-byte per packet bandwidth budget when compared to the ideal Aggregation method.

V. Related Work

Heavy hitters are an active research area on both intervals [12], [42], [30], [14] and sliding windows [13], [12]. HH integration in the single-device mode is a research challenge.

NetQRE [45] allows the network administrator to write a measurement program. The program can describe HH and HHH as well as sliding windows. However, their algorithm is exact rather than approximate and requires a space that is linear in the window size which is expensive for large windows.

Hierarchical heavy hitters. The HHH problem was introduced by [18], which also presented the first algorithm. The problem then attracted a large body of follow-up work as well as an extension to multiple dimensions [33], [11], [17]. MST [33] is a conceptually simple multidimensional HHH algorithm that uses multiple independent HH instances; one instance is used for each prefix pattern. Upon a packet arrival, all instances are updated with their corresponding prefixes. The set of hierarchical heavy hitters is then calculated from the set of (plain) heavy hitters of each prefix type. The algorithm requires $O\left(\frac{H}{\epsilon}\right)$ space and $O\left(H\right)$ update time. Memento and H-Memento are the first sliding window algorithms that leverage a sampling-like approach for additional speed. As such, they are considerably faster but are (slightly) less accurate.

Network-wide measurement. The problem of network-wide measurement is becoming increasingly popular [22], [44], [43], [29], [9]. A centralized controller collects data from all measurement points to form a network-wide perspective. Measurement points are placed in the network so that each packet is measured only once. The work of [6] suggests marking monitored packets and allows for more flexible measurement point placement. Our work diverges from these approaches since we view network-wide measurements for sliding windows. As such, the trade-offs between accuracy and bandwidth that we experience are different than that of interval based measurements. For instance, our work shows that the Batch method is superior to the commonly used Aggregation method.

VI. Analysis

We analyze Memento and H-Memento in Section VI-A, and D-Memento and D-H-Memento in Section VI-B.

A. *Memento's and H-Memento's Analyses*

We focus on the correctness of H-Memento, as the correctness of Memento follows (one can view Memento as the special case of H-Memento, i.e., with $H = 1$). We prove that the HHH set returned by H-Memento satisfies the accuracy and coverage properties (see Definition 6).

We model the H-Memento's update procedure as a balls and bins experiment where we select one out of H prefixes and then update that prefix with probability τ. For simplicity, we assume that $\frac{H}{\tau} \in \mathbb{N}$. Thus, we have $V \triangleq \frac{H}{\tau}$ bins and W balls. Upon a packet arrival, we place a ball in one of the bins; if the bin is one of the first H, we perform a full update for the sampled prefix; otherwise, we perform a window update. Definition 8 formulates this model. Alternatively, Memento is modeled as the degenerate case where $|H| = 1$ and we update the fully specified prefix.

Definition 8. *For each bin (i) and set of packets (K), denote by X_i^K the number of balls (from K) in bin i. When the set K contains all packets, we use the notation X_i.*

We require confidence intervals for any X_i and a set K. However, the X_i's are correlated as $\sum_{i=1}^{V} X_i = W$ and therefore we use the technique of Poisson approximation. It enables us to compute confidence intervals for *independent* Poisson variables $\{Y_i\}$ and convert back to the balls and bins case. Formally, let $Y_1^K, ..., Y_V^K \sim Poisson\left(\frac{K}{V}\right)$, be **independent** variables representing the number of balls in each bin. We now use Lemma 1 to get intervals for the X_i's.

Lemma 1 (Corollary 5.11, page 103 of [34]). *Let $\mathfrak{E}$ be an event whose probability monotonically increases with the number of balls. If the probability of $\mathfrak{E}$ is p in the Poisson case, then it is at most $2p$ in the exact case.*

Accuracy Analysis: To prove accuracy, we show that, for every prefix (p): $\Pr\left(\left|f_p^W - \widehat{f_p^W}\right| \leq \varepsilon W\right) \geq 1-\delta$. We have multiple sources of error and thus we first quantify the sampling error. Let Y_i^p be the Poisson variable corresponding to a prefix p. That is, the set K contains all the packets that are generalized by p. Therefore: $E(Y_i^p) = \frac{f_p^W}{V}$. We need to show that: $\Pr\left(|Y_i^p - E(Y_i^p)| \leq \epsilon_s \frac{W}{V}\right) \geq 1-\delta$. Confidence intervals for Poisson variables are well studied [38], we use Lemma 2.

Lemma 2 (Proved in [40])**.** *Let Y be a Poisson random variable. Then $\Pr\left(|Y - E(Y)| \geq Z_{1-\delta}\sqrt{E(Y)}\right) \leq \delta$; here, Z_α is the z value that satisfies $\Phi(z) = \alpha$ and $\Phi(z)$ is the cumulative density of the normal distribution with mean 0 and standard deviation of 1.*

Lemma 2 lays the groundwork for our main accuracy result.

Theorem 1. $\Pr\left(\left|X_i{}^pV - f_p^W\right| \geq \varepsilon_s W\right) \leq \delta_s$. *for* $\tau \geq Z^2_{1-\frac{\delta}{2}} HW^{-1}\varepsilon_s{}^{-2}$

Proof. Use Lem. 2 to get: $\Pr\left(\left|Y_i{}^p - \frac{f_p^W}{V}\right| \geq Z_{1-\frac{\delta_s}{2}}\sqrt{\frac{f_p^W}{V}}\right) \leq \frac{\delta_s}{2}$. Since we do not know the exact value of f_p^W, we assert that $f_p^W \leq W$ to get: $\Pr\left(\left|Y_i{}^p - \frac{f_p^W}{V}\right| \geq Z_{1-\frac{\delta_s}{2}}\sqrt{\frac{W}{V}}\right) \leq \frac{\delta_s}{2}$. We need error of the form: $\frac{\epsilon_s \cdot W}{V}$ and thus set: $\frac{\epsilon_s \cdot W}{V} = Z_{1-\frac{\delta_s}{2}}\sqrt{\frac{W}{V}} = Z_{1-\frac{\delta_s}{2}}\sqrt{\frac{W\tau}{H}}$ We extract τ to get: $\tau \geq Z^2_{1-\frac{\delta_s}{2}} \frac{H}{W}\varepsilon_s{}^{-2}$. Thus, when $\tau \geq Z^2_{1-\frac{\delta}{2}} HW^{-1}\varepsilon_s{}^{-2}$, we have that: $\Pr\left(\left|Y_i{}^p - \frac{f_p^W}{V}\right| \geq \frac{\varepsilon_s W}{V}\right) \leq \frac{\delta_s}{2}$. We multiply by V and get: $\Pr\left(\left|Y_i{}^pV - f_p^W\right| \geq \varepsilon_s W\right) \leq \frac{\delta_s}{2}$. Finally, since Y_i^p is monotonically increasing with the number of balls (f_p^W), use Lemma 1 to conclude: $\Pr\left(\left|X_i{}^pV - f_p^W\right| \geq \varepsilon_s W\right) \leq \delta_s$. □

To reduce clutter, we denote $\psi \triangleq Z^2_{1-\frac{\delta}{2}} HW^{-1}\varepsilon_s{}^{-2}$. Theorem 1 shows that when $\tau \geq \psi$ the sample is accurate enough. The error of the underlying Memento algorithm is proportional to the number of *sampled* packets. We compensate for fluctuations in sample size by allocating (slightly) more counters as explained in Corollary 1.

Corollary 1. *Consider the number of updates (from the last W items) to the underlying algorithm (X). If $\tau \geq \psi$, then $\Pr\left(X \leq \frac{W}{V}(1+\varepsilon_s)\right) \geq 1-\delta_s$.*

Proof. Theorem 1 yields: $\Pr\left(\left|X - \frac{W}{V}\right| \geq \varepsilon_s W\right) \leq \delta_s$. Thus: $\Pr\left(X \leq \frac{W}{V}(1+\varepsilon_s)\right) \geq 1-\delta_s$. □

Corollary 1 means that, by allocating slightly more space to the underlying algorithm, we can compensate for possible oversampling. Generally, we configure an algorithm $(\mathbb{A})$ that solves $(\varepsilon_a, \delta_a)$ - WINDOW FREQUENCY ESTIMATION with $\varepsilon'_a \triangleq \frac{\varepsilon_a}{1+\varepsilon_s}$. Applying Corollary 1, we get that, with probability $1-\delta_s$, there are at most $(1+\varepsilon_s)\frac{W}{V}$ sampled packets. Using the union bound we have that with probability $1-\delta_a-\delta_s$: $\left|X^p - \widehat{X^p}\right| \leq \varepsilon_{a'}(1+\varepsilon_s)\frac{W}{V} = \frac{\varepsilon_a(1+\varepsilon_s)}{1+\varepsilon_s}\frac{W}{V} = \varepsilon_a\frac{W}{V}$. For example, WCSS requires $4,000$ counters for $\epsilon_a = 0.001$. If we set $\epsilon_s = 0.001$, we now require 4004 counters. Hereafter, we assume that the algorithm is already configured to accommodate this problem.

Theorem 2. *Consider an algorithm $(\mathbb{A})$ that solves the (ϵ_a, δ_a) -WINDOW FREQUENCY ESTIMATION problem. If $\tau \geq \psi$, then for $\delta \geq \delta_a + 2\cdot\delta_s$ and $\epsilon \geq \epsilon_a + \epsilon_s$, $\mathbb{A}$ solves (ϵ, δ) - WINDOW FREQUENCY ESTIMATION.*

Proof. We employ Theorem 1. That is, we have that: $\Pr\left[\left|f_p^W - X_pV\right| \geq \varepsilon_s W\right] \leq \delta_s$. (1)
$\mathbb{A}$ solves (ϵ_a, δ_a) - WINDOW FREQUENCY ESTIMATION and provides us with an estimator $\widehat{X^p}$ for X^p – the number of updates for a prefix p. According to Corollary 1: $\Pr\left(\left|X^p - \widehat{X^p}\right| \leq \frac{\varepsilon_a W}{V}\right) \geq 1-\delta_a-\delta_s$. Multiplying by V yields: $\Pr\left(\left|X^pV - \widehat{X^p}V\right| \geq \varepsilon_a W\right) \leq \delta_a + \delta_s$. (2)
We need to show that: $\Pr\left(\left|f_p^W - \widehat{X^p}V\right| \leq \varepsilon W\right) \geq 1-\delta$. Note that: $f_p^W = E(X^p)V$ and $\widehat{f_p^W} \triangleq \widehat{X^p}V$. Thus,

$$\Pr\left(\left|f_p^W - \widehat{f_p^W}\right| \geq \varepsilon W\right) = \Pr\left(\left|f_p^W - \widehat{X^p}V\right| \geq \varepsilon W\right) = \tag{3}$$
$$\Pr\left(\left|f_p^W + (X^pV - X^pV) - V\widehat{X^p}\right| \geq (\epsilon_a+\epsilon_s)W\right) \leq \tag{4}$$
$$\Pr\left(\left[\left|f_p^W - X^pV\right| \geq \varepsilon_s W\right] \vee \left[\left|X^pV - \widehat{X^p}V\right| \geq \varepsilon_a W\right]\right).$$

The last inequality follows from the observation that if the error of (3) exceeds ϵW, then one of the events occurs. We bound this expression with the Union bound.
$\Pr\left(\left|f_p^W - \widehat{f_p^W}\right| \geq \varepsilon W\right) \leq \Pr\left(\left|f_p^W - X^pV\right| \geq \varepsilon_s W\right) +$
$\Pr\left(\left|X^pV - \widehat{X^p}H\right| \geq \varepsilon_a W\right) \leq \delta_a + 2\delta_s,$
where the last inequality follows from Eq. (1) and (2). □

Theorem 2 implies accuracy, as it guarantees that, with probability $1-\delta$, the estimated frequency of *any* prefix is within εW of its real frequency. In particular, this means that the HHH prefix estimations are within εW bound as shown by Corollary 2.

Corollary 2. *If $\tau \geq \psi$, then H-Memento satisfies accuracy for $\delta = \delta_a + 2\delta_s$ and $\epsilon = \epsilon_a + \epsilon_s$.*

Furthermore, by considering the degenerate case where we always select fully specified items (*i.e.,* $H = 1$ and $V = \tau^{-1}$), we conclude the correctness of Memento, as stated in the following Corollary 3.

Corollary 3. *If $\tau \geq Z^2_{1-\frac{\delta_s}{2}} W^{-1}\epsilon_s^{-2}$ then Memento solves the (ϵ, δ) - WINDOW FREQUENCY ESTIMATION problem for $\delta = 2\cdot\delta_s$ and $\varepsilon = \varepsilon_a + \varepsilon_s$.*

Coverage Analysis: We now show that H-Memento satisfies the coverage property (Definition 6). That is, $\Pr\left(\widehat{C_{q|P}} \geq C_{q|P}\right) \geq 1-\delta$. Conditioned frequencies are calculated differently for 1D and 2D, thus Section VI-A1 shows coverage for 1D and Section VI-A2 for 2D.

1) 1D: Lemma 3 shows an expression for $C_{q|P}$.

Lemma 3 (Proved in [33])**.** *In one dimension: $C_{q|P} = f_q^W - \sum_{h\in G(q|P)} f_h^W$.*

We use Lemma 3 to show that the estimations of Algorithm 2 are conservative.

Lemma 4. *The conditioned frequency estimation of Algorithm 2 is: $\widehat{C_{q|P}} = \widehat{f_q^W}^{+} - \sum_{h\in G(q|P)} \widehat{f_h^W}^{-} + 2Z_{1-\delta}\sqrt{WV}$.*

Proof. Looking at Line 7 in Algorithm 2, we get that: $\widehat{C_{q|P}} = \widehat{f_q^W}^+ + calcPred(q,P)$. That is, we need to verify that the return value $calcPred(q,P)$ in one dimension (Algorithm 3) is $\sum_{h\in G(q|P)} \widehat{f_h^W}^-$. Finally, the addition of $2Z_{1-\delta}\sqrt{WV}$ is done in line 8. ☐

Theorem 3. $\Pr\left(\widehat{C_{q|P}} \geq C_{q|P}\right) \geq 1-\delta$.

Proof. From Lemma 4, we have $\widehat{C_{q|P}} = \widehat{f_q^W}^+ - \sum\limits_{h\in G(q|P)} \widehat{f_h^W}^- + 2Z_{1-\frac{\delta}{8}}\sqrt{WV}$. It is enough to show that the randomness is bounded by $2Z_{1-\frac{\delta}{8}}\sqrt{WV}$ with probability $1-\delta$ as $\widehat{f_p^W}^+ \geq f_p^W$ and $f_h^W \leq \widehat{f_h^W}^-$. We denote by K the set of packets that affect the calculation of $\widehat{C_{q|P}}$. We split K into two: K^+ contains packets that increase the value of $\widehat{C_{q|P}}$ and K^- contains these that decrease it. We use K^+ to estimate the sample error in $\widehat{f_q^W}$ and K^- for estimating the error in $\sum\limits_{h\in G(q|P)} \widehat{f_h^W}^-$.

We denote by Y^{K^+} the number of balls in the positive sum and use Lemma 2. $C_{q|p}$ is non-negative. Thus $E\left(Y^{K^-}\right) \leq \frac{W}{V}$and $\Pr\left(\left|Y^{K^+} - E\left(Y^{K^+}\right)\right| \geq Z_{1-\frac{\delta}{8}}\sqrt{\frac{W}{V}}\right) \leq \frac{\delta}{4}$. Similarly, we use Lemma 2 to bound the error of Y^{K^-}. $\Pr\left(\left|Y^{K^-} - E\left(Y^{K^-}\right)\right| \geq Z_{1-\frac{\delta}{8}}\sqrt{\frac{W}{V}}\right) \leq \frac{\delta}{4}$.
Y^{K^+} and Y^{K^-} are monotonic with the number of balls. We apply Lemma 1 and use the Union bound to conclude: $\Pr\left(\widehat{C_{q|P}} \geq C_{q|P}\right) \leq 2\Pr\left(H\left(Y^{K^-} + Y^{K^+}\right) \geq VE\left(Y^{K^-}+Y^{K^+}\right) + 2Z_{1-\frac{\delta}{8}}\sqrt{NV}\right) \leq 1-2\frac{\delta}{2} = 1-\delta$. ☐

Theorem 4. *Algorithm 2 solves* $(\delta,\varepsilon,\theta)$ - APPROXIMATE WINDOW HHH *for* $\tau \geq \psi$, $\delta = \delta_a + 2\delta_s$, $\varepsilon = \varepsilon_s + \varepsilon_a$.

Proof. We need to prove that Algorithm 2 satisfies both accuracy and coverage. Corollary 2 shows accuracy, while Theorem 3 says that: $\Pr\left(\widehat{C_{q|P}} \geq C_{q|P}\right) \geq 1-\delta$.

Consider a prefix q such that $q \notin P$, where P is the set of HHH. We know that $\widehat{C_{q|P}} < \theta W$ because otherwise q would have been an HHH prefix. Thus, with probability $1-\delta$, we get: $C_{q|P} < \widehat{C_{q|P}} < \theta W$, which implies that $\Pr\left(C_{q|P} < \theta W\right) \geq 1-\delta$ and hence Algorithm 2 satisfies coverage as well. ☐

2) 2D: Next, we establish that H-Memento is correct for two dimensions. To that end, we introduce the notation of $G(q|P)$, which stands for the closest prefixes to q in set P.

Definition 9 (Best generalization)**.** *Define* $G(q|P)$ *as the set* $\{p : p \in P, q \prec p, \neg\exists p' \in P : q \prec p' \prec p\}$. *Intuitively,* $G(q|P)$ *contains the prefixes that are the closest ancestors (from* P*) of* q*. Thus,* q *does not generalize any prefix that generalizes a prefix in* $G(q|P)$*.*

Lemma 5 quantifies 2D conditioned frequencies.

Lemma 5 (Proved in [33])**.** *In two dimensions,* $C_{q|P} = f_q^W - \sum\limits_{h\in G(q|P)} f_h^W + \sum\limits_{h,h'\in G(q|P)} f_{\text{glb}(h,h')}^W$

Next, Lemma 6 formalizes the expression Algorithm 2 uses to calculate conditioned frequencies in two dimensions. Then, Theorem 5 lays the groundwork for coverage.

Lemma 6. *In two dimensions, Algorithm 2 calculates the conditioned frequency as:* $\widehat{C_{q|P}} = \widehat{f_q^W}^+ - \sum\limits_{h\in G(q|P)} \widehat{f_h^W}^- + \sum\limits_{h,h'\in G(q|P)} \widehat{f_{\text{glb}(h,h')}^W}^+ + 2Z_{1-\frac{\delta}{8}}\sqrt{WV}$.

Proof. The proof follows from Algorithm 2. Line 7 adds $\widehat{f_q^+}$ while Line 8 is responsible for the last element $(2Z_{1-\frac{\delta}{8}}\sqrt{WV})$. The rest is from the calcPredecessors method in Algorithm 4. ☐

Theorem 5. $\Pr\left(\widehat{C_{q|P}} \geq C_{q|P}\right) \geq 1-\delta$.

Proof. Observe Lemma 5 and notice that if there is no sampling error: $\widehat{f_q^W}^+ - \sum\limits_{h\in G(q|P)} \widehat{f_h^W}^- + \sum\limits_{h,h'\in G(q|P)} \widehat{f^W}^+_{\text{glb}(h,h')}$ is a conservative estimate. Thus, we now show that this error is less than $2Z_{1-\frac{\delta}{8}}\sqrt{WV}$ with probability $1-\delta$. We denote by K the packets that affect $C_{q|P}$ and since the expression of $\widehat{C_{q|P}}$ is not monotonic. As before, we split in two: K^+ are packets that increase $\widehat{C_{q|P}}$, while and K^- decrease it. Similarly, we denote by $\{Y_i^K\}$ the number of packets from K in bin i of the Poisson model. We also denote the random variable Y^{K^+} that counts *how many* balls from K had increased $\widehat{C_{q|P}}$. Lemma 2 binds Y^{K^+} in the following manner: $\Pr\left(\left|Y^{K^+} - E\left(Y^{K^+}\right)\right| \geq Z_{1-\frac{\delta}{8}}\sqrt{\frac{W}{V}}\right) \leq \frac{\delta}{4}$.
Similarly, we denote by Y^{K^-} the number of packets from K with negative impact on $\widehat{C_{q|P}}$. Using Lemma 2 results in: $\Pr\left(\left|Y^{K^-} - E\left(Y^{K^-}\right)\right| \geq Z_{1-\frac{\delta}{8}}\sqrt{\frac{W}{V}}\right) \leq \frac{\delta}{4}$. Y^{K^+} and Y^{K^-} are monotonic with the number of balls. Thus, we apply Lemma 1 and conclude: $\Pr\left(\widehat{C_{q|P}} \geq C_{q|P}\right) \leq 2\Pr\left(V\left(Y^{K^-}+Y^{K^+}\right) \geq \left(VE\left(Y^{K^-}+Y^{K^+}\right) + 2Z_{1-\frac{\delta}{8}}\sqrt{WV}\right)\right) \leq 1-2\frac{\delta}{2} = 1-\delta$. ☐

3) Putting It All Together:

Corollary 4. *If* $\tau > \psi$ *then H-Memento satisfies coverage. That is, let* P *be the HHH set of H-Memento; given a prefix* $q \notin P$ *we have* $\Pr\left(C_{q|P} < \theta W\right) > 1-\delta$.

Proof. Theorem 3 shows coverage in one dimension and Theorem 5 in two. These theorems guarantee that: $\Pr\left(C_{q|P} < \widehat{C_{q|P}}\right) > 1-\delta$. Let $q \notin P$, which means that $\widehat{C_{q|P}} < \theta W$. However, with probability $1-\delta$, $C_{q|P} < \widehat{C_{q|P}} < \theta W$ and therefore $C_{q|P} < \theta W$ as well. ☐

H-Memento's analysis conclusion.

Theorem 6. *H-Memento solves* (δ,ϵ,θ) - APPROXIMATE WINDOW HIERARCHICAL HEAVY HITTERS *for* $\tau \geq Z^2_{1-\frac{\delta}{2}} HW^{-1}{\varepsilon_s}^{-2}$.

Proof. H-Memento provides both accuracy by Corollary 2 and coverage by Corollary 4 (see definition 6). ☐

Note that H-Memento is correct when $\tau > \psi$. That is, larger windows (W), or larger ϵ_s, allow for more aggressive sampling. Finally, Memento and H-Memento perform updates in constant time and that H-Memento requires $O(H/\varepsilon)$ space.

B. D-Memento and D-H-Memento Analysis

Intuitively, the error in D-Memento and D-H-Memento comes from two origins. First, there is the error due to *sampling*, which is quantified by Corollary 3 and Theorem 6. However, there is an additional error that is caused by the *delay in transmission*, as the measurement points only send the sampled packets once in every $b\tau^{-1}$ packets. If a measurement point has a low traffic rate, it may take a long time for it to see $b\tau^{-1}$ packets; in this case, all of its samples may be obsolete and may not belong in the most recent window. Therefore, our first step is to reason about the delayed reporting error.

Notations and definitions. We denote the *bandwidth budget* as $\mathcal{B}$ bytes/packet. This communication is done using standard packets, which have header field overheads. We denote by $\mathbb{O}$ the minimal header size (in bytes) of the chosen transmission protocol (*e.g.,* 64 bytes for TCP). Next, reporting a sampled packet requires E bytes (*e.g.,* 4 bytes for srcip, or 8 bytes for (srcip,dstip) pair). We also denote by m the total number of measurement points.

Model. Intuitively, we can choose two (dependent) parameters: the sampling rate, τ, and the batch size b. That is, each measurement point samples with probability τ until it gathers b packets. At this point, it assembles an $(\mathbb{O}+Eb)$-sized packet that encodes the sampled packet and sends it to the controller. As the expected number of packets required to gather a b sized batch is $b\tau^{-1}$, the bandwidth constraint can be written as $(\mathbb{O}+Eb)/(b\tau^{-1}) \leq \mathcal{B}$. Specifically, this allows to express the maximum allowed sampling probability as $\tau = \mathcal{B}b/(\mathbb{O}+Eb)$ since sampling at a lower rate would not utilize the entire bandwidth and would result in sub-optimal accuracy.

Accuracy of the Batch and Sample methods. We can now quantify the error of the Batch and Sample methods. Intuitively, we have to factor the delays in communication (as we only report per a fixed number of packets to stay within the bandwidth budget). For example, if there are two measurement points in which one processes a million requests per second while the other only a thousand, the batches of the second point would include many obsolete packets that are not within the current window. However, recall that these reports only reflect $b\tau^{-1}$ packets at each of the m points. Thus:

Theorem 7. *Batch method's delayed reporting error is bounded by* $mb\tau^{-1}$.

Next, Corollary 3 and Theorem 6 enable us to bound the sampling error as a function of τ, while Theorem 7 bounds the delayed reporting error. The following theorem applies for D-Memento (using $H = 1$) and for D-H-Memento (using the appropriate H value). As the round trip time inside the data center is small compared to window sizes that are of interest, the error caused by the delay of packet transmissions is negligible, and thus we do not factor it here. Theorem 8 quantifies the overall error in the Batch method; the error of the Sample method is derived when setting $b = 1$.

Theorem 8. *Given overhead $\mathbb{O}$, batch size b, bandwidth budget $\mathcal{B}$, sample payload size S, window size W and confidence δ_s, the overall error $\mathfrak{E}_b$ (in packets) is at most:*

$$\mathfrak{E}_b = m(\mathbb{O}+Eb)/\mathcal{B} + \sqrt{HWZ_{1-\frac{\delta_s}{2}}(\mathbb{O}+Eb)/(\mathcal{B}b)}.$$

Proof. According to Theorem 6, we have that $\epsilon_s = \sqrt{HW^{-1}Z_{1-\frac{\delta_s}{2}}\tau^{-1}}$. This means that our overall error is bounded by: $\mathfrak{E}_b = bm\tau^{-1} + W\epsilon_s = bm\tau^{-1} + \sqrt{HWZ_{1-\frac{\delta_s}{2}}\tau^{-1}} = m(\mathbb{O}+Eb)/\mathcal{B} + \sqrt{HWZ_{1-\frac{\delta_s}{2}}(\mathbb{O}+Eb)/(\mathcal{B}b)}$. $\square$

Formally, we showed a bound of $\mathfrak{E}_b$, for each choice of b. The guarantees for the Sample method are given by fixing $b = 1$. The next step is to use Theorem 8 to calculate the optimal batch size b given a bandwidth budget $\mathcal{B}$. Thus, we get the best achievable accuracy for the Batch method within the bandwidth limitation. We have: $\frac{\partial \mathfrak{E}_b}{\partial b} = mE/\mathcal{B} + \frac{HWZ_{1-\frac{\delta_s}{2}}(E/B-\mathbb{O}/(\mathcal{B}b^2))}{2\sqrt{(\mathbb{O}+Eb)/(\mathcal{B}b)}}$. We then compare this expression to zero to compute the optimal batch size b. This is easily done with numerical methods.

For example, for a TCP connection ($\mathbb{O} = 64$); ten measurement points ($k = 10$); source IP hierarchy ($E = 4, H = 5$); error probability of $\delta = 0.01\%$; a window of size $W = 10^6$; and a bandwidth quota of $\mathcal{B} = 1$ byte per packet, the optimal batch size is $b = 44$. The resulting (overall) error guarantee is 13K packets (i.e., an error of 1.3%). Increasing the bandwidth budget to $\mathcal{B} = 5$ bytes decreases the absolute error to 5.3K packets (0.53%) while increasing the optimal batch size to $b = 68$. When increasing the window size (W), the *absolute* error increases by an $O\left(\sqrt{W}\right)$ factor and the error (as a fraction of W) decreases. For example, increasing the window size to 10^7 increases the optimal batch size to $b = 109$, while reducing the error to 0.15%. Alternatively, 2D source/destination hierarchies (increasing H from 5 to 25) result in a slightly larger error and a higher optimal batch size.

Figure 11 illustrates the accuracy guarantee provided by each method. We compare three synchronization variants – Sample, Batch with $b = 100$, and Batch with an optimal b (varies with $\mathcal{B}$), as explained above. As depicted, Sample has the smallest delay error and yet provides the worst guarantees as it conveys little information within the bandwidth budget. The 100-Batch method has lower a sampling error (as its sampling rate is higher), but its reporting delay makes the overall error larger. For larger values of $\mathcal{B}$, the optimal batch size grows closer to 100 and the accuracy gap narrows.

VII. Conclusions

Our study highlights the potential benefits of sliding-window measurements and makes them practical for network applications. Specifically, we showed that window-based measurements detect traffic changes faster, and thus enable more agile applications. Existing window algorithms are slow and and do not provide a network-wide view. Our

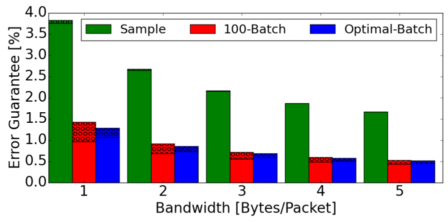


Fig. 11: Comparing the accuracy *guarantees* of varying synchronization techniques. The parts hatched with circles quantify the bound on the error that is caused by the delayed synchronization.

window algorithms have higher throughput than existing algorithms and thus would require fewer resources to implement them (although existing algorithms can also potentially be used with parallelism) Accordingly, we introduced the Memento family of HH and HHH algorithms for both single-device and network-wide measurements. We analyzed the algorithms and extensively evaluated them on real traffic traces. Our evaluations indicate that the Memento algorithms meet the necessary speed and efficiently to provide network-wide visibility. Therefore, our study makes sliding-window HH and HHH measurements be a practical option for the next generation of network applications. We note that while our HHH algorithm processes packets faster than previous art, it does not improve their query time. We believe that real-time identification of HHH is a promising future research direction.

We open-sourced the Memento algorithms and the HAProxy load-balancer extension that provides capabilities to block and rate-limit traffic from entire sub-networks (rather than from individual flows) [3].

References

[1] Cloudflare. What is an HTTP flood DDoS attack?

[2] Imperva Inc, Learning Center. What is an HTTP flood attack.

[3] Memento algorithms code and HAProxy extension. https://github.com/DHMementoz/Memento.

[4] NETSCOUT. What is an HTTP Flooding DDoS Attack?

[5] see http://www.lasr.cs.ucla.edu/ddos/traces/.

[6] Y. Afek, A. Bremler-Barr, S. L. Feibish, and L. Schiff. Detecting heavy flows in the SDN match and action model. *Computer Networks*, 136:1 – 12, 2018.

[7] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: Minimal Near-optimal Datacenter Transport. *ACM SIGCOMM*, pages 435–446, 2013.

[8] D. Anderson, P. Bevan, K. Lang, E. Liberty, L. Rhodes, and J. Thaler. A high-performance algorithm for identifying frequent items in data streams. In *ACM Internet Measurement Conference*, pages 268–282, 2017.

[9] R. B. Basat, G. Einziger, S. L. Feibish, J. Moraney, and D. Raz. Network-wide routing-oblivious heavy hitters. *ACM ANCS*, 2018.

[10] R. B. Basat, G. Einziger, R. Friedman, M. C. Luizelli, and E. Waisbard. Constant time updates in hierarchical heavy hitters. *ACM SIGCOMM*, 2017.

[11] R. B. Basat, G. Einziger, R. Friedman, M. C. Luizelli, and E. Waisbard. Volumetric hierarchical heavy hitters. In *IEEE MASCOTS*, 2018.

[12] R. Ben Basat, G. Einziger, and R. Friedman. Fast flow volume estimation. In *ICDCN*, 2018.

[13] R. Ben-Basat, G. Einziger, R. Friedman, and Y. Kassner. Heavy Hitters in Streams and Sliding Windows. In *IEEE INFOCOM*, 2016.

[14] R. Ben-Basat, G. Einziger, R. Friedman, and Y. Kassner. Randomized admission policy for efficient top-k and frequency estimation. In *IEEE INFOCOM*, 2017.

[15] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild (univ 1 dataset). In *ACM Internet Measurement Conference*, 2010.

[16] M. Chiesa, G. Rétvári, and M. Schapira. Lying your way to better traffic engineering. In *ACM CoNEXT*, 2016.

[17] K. Cho. Recursive lattice search: Hierarchical heavy hitters revisited. In *ACM IMC*, page 283289, 2017.

[18] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Finding Hierarchical Heavy Hitters in Data Streams. In *VLDB*, 2003.

[19] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Diamond in the Rough: Finding Hierarchical Heavy Hitters in Multi-dimensional Data. In *SIGMOD*, pages 155–166, 2004.

[20] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Finding Hierarchical Heavy Hitters in Streaming Data. *ACM Trans. Knowl. Discov. Data*, 1(4):2:1–2:48, Feb. 2008.

[21] G. Einziger, M. C. Luizelli, and E. Waisbard. Constant time weighted frequency estimation for virtual network functionalities. In *ICCCN*, pages 1–9, July 2017.

[22] R. Harrison, Q. Cai, A. Gupta, and J. Rexford. Network-wide heavy hitter detection with commodity switches. In *ACM SOSR*, pages 8:1–8:7, 2018.

[23] P. Hick. CAIDA Anonymized 2016 Internet Trace, equinix-chicago 2016-02-18 13:00-13:05 UTC, Direction A.

[24] S. Hilton. Dyn Analysis Summary Of Friday October 21 Attack. Available: https://dyn.com/blog/dyn-analysis-summary-of-friday-october-21-attack/.

[25] R. Y. S. Hung and H. F. Ting. Finding heavy hitters over the sliding window of a weighted data stream. In *LATIN*, 2008.

[26] N. Katta, A. Ghag, M. Hira, I. Keslassy, A. Bergman, C. Kim, and J. Rexford. Clove: Congestion-aware load-balancing at the virtual edge. In *ACM CoNEXT*, 2017.

[27] A. Khalimonenko, O. Kupreev, and E. Badovskaya. DDoS attacks in Q1 2018. Available: https://securelist.com/ddos-report-in-q1-2018/85373/.

[28] L. K. Lee and H. F. Ting. A simpler and more efficient deterministic scheme for finding frequent items over sliding windows. In *ACM PODS*, 2006.

[29] Y. Li, R. Miao, C. Kim, and M. Yu. FlowRadar: A better NetFlow for data centers. In *Usenix NSDI*, 2016.

[30] A. Metwally, D. Agrawal, and A. E. Abbadi. Efficient Computation of Frequent and Top-k Elements in Data Streams. In *ICDT*, 2005.

[31] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *ACM SIGCOMM*, pages 15–28, 2017.

[32] M. Mitzenmacher, T. Steinke, and J. Thaler. Hierarchical heavy hitters with the space saving algorithm. *CoRR*, 2011. Conference version appeared in ALENEX 2012.

[33] M. Mitzenmacher, T. Steinke, and J. Thaler. Hierarchical Heavy Hitters with the Space Saving Algorithm. In *ALENEX*, 2012.

[34] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[35] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. DREAM: Dynamic resource allocation for software-defined measurement. In *ACM SIGCOMM*, 2014.

[36] S. Muthukrishnan et al. Data streams: Algorithms and applications. *Foundations and Trends® in Theoretical CS*, 2005.

[37] K. Nyalkalkar, S. Sinhay, M. Bailey, and F. Jahanian. A comparative study of two network-based anomaly detection methods. In *IEEE Infocom*, 2011.

[38] V. Patil and H. Kulkarni. Comparison of confidence intervals for the poisson mean: Some new aspects. *R. Stat. J.*, 2012.

[39] R. Schweller, A. Gupta, E. Parsons, and Y. Chen. Reversible sketches for efficient and accurate change detection over network data streams. In *ACM IMC*, 2004.

[40] N. C. Schwertman and R. A. Martinez. Approximate poisson confidence limits. *Comm. in Stat. Theory and Methods, 1994.*

[41] V. Sekar, N. G. Duffield, O. Spatscheck, J. E. van der Merwe, and H. Zhang. Lads: Large-scale automated ddos detection system. In *USENIX ATC*, 2006.

[42] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford. Heavy-hitter detection entirely in the data plane. In *ACM SOSR*, 2017.

[43] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig. Adaptive measurements using one elastic sketch. *IEEE/ACM Transactions on Networking*, 27(6):2236–2251, Dec 2019.

[44] T. Yang, J. Jiang, P. Liu, J. G. Qun Huang, Y. Zhou, R. Miao, X. Li, and S. Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. *ACM SIGCOMM*, 2018.

[45] Y. Yuan, D. Lin, A. Mishra, S. Marwaha, R. Alur, and B. T. Loo. Quantitative network monitoring with NetQRE. In *ACM SIGCOMM*, pages 99–112, 2017.